INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Programming with Junior

Frédéric Boussinot — Laurent Hazard — Jean-Ferdy Susini

N° 4027

Octobre 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4027--FR

Programming with Junior*

Frédéric Boussinot† , Laurent Hazard‡ , Jean-Ferdy Susini§

Thème 1 — Réseaux et systèmes
Projet Mimosa

Rapport de recherche n° 4027 — Octobre 2000 — 31 pages

Abstract: Junior is a Java-based language for programming reactive behaviors. It provides programmers with concurrency, broadcast events, and several primitives for gaining fine control over reactive programs executions. It is used through an API named **Jr** which masks several existing implementations.

Key-words: Reactive Programming, Concurrency, Java

* With support from France-Telecom R&D, and from EC (IST 99-11488 PING Project)
† EMP/CMA-INRIA
‡ FT-R&D
§ EMP/CMA-INRIA

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : 04 92 38 77 77 - International : +33 4 92 38 77 77 —Fax : 04 92 38 77 65 - International : +33 4 92 38 77 65

La programmation en Junior

Résumé : **Junior** est un langage construit au dessus de Java pour programmer des comportements réactifs. Il autorise la concurrence, les événements diffusés et définit plusieurs primitives permettant d'obtenir un contrôle fin sur l'exécution des programmes réactifs. Il est utilisé au travers d'une API, nommée **Jr**, qui masque les implémentations existantes.

Mots-clés : Programmation réactive, Parallélisme, Concurrence, Java

1 Introduction

Junior [7] is a Java-based language for programming reactive behaviors. It is closely related to SugarCubes[3] and is actually a descendant of it. The aim of this paper is to describe how to use **Junior**.

The reactive approach proposes a flexible paradigm for programming reactive systems[6], especially those which are dynamic (that is, the number of components and their connections can change during execution). Reactive programming provides programmers with concurrency, broadcast events, and several primitives for gaining fine control over reactive programs executions. At the basis of reactive programming is the notion of a reaction: reactive programs are reacting to activations issued from the external world. Program reactions are often called *instants*. The two main notions are reactive instructions whose semantics refer to instants, and reactive machines whose purpose is to execute reactive instructions in an environment made of instantaneously broadcast events.

Basically, programming with **Junior** means:

- writing a *reactive instruction*, which describes an application program;
- declaring a *reactive machine*, to run the program;
- dropping the program into the machine;
- running the machine; this is usually performed using a non terminating loop, which cyclically makes the machine and the program *react*.

Programming in **Junior** has a dynamic aspect: machine programs can be augmented by new reactive instructions added during machine execution. New instructions dropped into a machine do not have to wait for the termination of the actual program, but are run *concurrently* with it.

Junior concurrent reactive instructions can communicate using *broadcast events* that are processed by reactive machines. Broadcasting is a powerful and fully modular means for communication and synchronization of concurrent components. Broadcasting in **Junior** has a special coherency property: during a machine reaction, the same event cannot be tested both present and absent, even by two distinct concurrent instructions.

Junior defines primitive constructs allowing for code (reactive instruction) migration over the network.

Junior is pure Java. It is provided with an API named **Jr**. Using **Jr**, programmers can define reactive instructions and reactive machines, and have possibility to run them. **Junior** is a programming language, defining constructs for reactive programming. It can also be seen as a Java programming framework. From this last point of view, **Junior** provides Java programmers with an alternative to the standard threading mechanism. The benefit is that **Junior** gives solutions to some well-known problems of Java threads (see [8] for a description of these problems).

The structure of the paper is as follows: First, section 2 gives an overview of **Junior**, describing its main characteristics. Then, the API for programming is given in section 3. Finally, one considers in section 4 several important specific points of **Junior** programming.

2 Junior Overview

This section introduces reactive instructions, reactive machines, and events, and it considers several related points, such as concurrency and migration. Finally, it discuss the relation with Java threads.

2.1 Reactive Instructions

Reactive instructions are state-based statements, run (one also says, *activated*) by reactive machines. Some cyclic instructions are never ending across instants, while others are reaching a final state after several activations; in this case, one says that they are *completely terminated*. Because states are embbeded in them, reactive instructions are not reentrant: they must be copied, in order to get new execution instances.

Reactive instructions are composed from a small set of basic constructors. For example, the constructor `Seq` puts two reactive instructions A and B in sequence: A is executed up to complete termination (remember that it may take several machine reactions), and then B is executed. The associated state of `Seq` encodes termination of the first component: the state changes when the first component completely terminates; then, following executions directly go to the second component, without considering the first one.

Reactive instructions are Java objects implementing the `Program` interface. They are built using static methods of class **Jr**[1]. For example, to define the sequence of two reactive instruction A, B, one writes:

```
Program p1 = Jr.Seq(A,B);
```

Syntax for constructors is very basic; for example, to put three instructions A, B, and C in sequence is not directly possible with one unique call of the `Seq` constructor, which must be called twice:

```
Program p2 = Jr.Seq(A,Jr.Seq(B,C));
```

This leads to a Lisp-like programming style, with a lot of parenthesis, which can be rather boring when writing large-sized reactive behaviors. To avoid this problem, more concise syntax have been proposed, which eventually translate into **Junior** programs; this point will be discussed later.

Among the reactive instructions are the ones, called *atoms*, used to interface with Java. Basically, an atom executes an action which possibly performs some interaction with the

[1] To simplify, one also calls "constructors" the static methods of **Jr** which call constructors of **Junior** classes.

Java environment. The action is executed once and the atom immediately terminates after first reaction. Execution of an atom is *atomic*: once started, execution of an atom always terminates without any interference with other atoms.

One special action is `Print`, defined for printing purposes; it is mainly used for tracing execution and it will be very useful in this presentation to get observable behaviors. For example, to print "`hello, world!`", one writes in **Junior**:

```
Jr.Atom(new Print("hello, world!"));
```

2.2 Reactive Machines

Two constructors are provided by **Jr** to build machines: `SyncMachine` and `UnSyncMachine` which both implements interface `Machine`. The two kinds of machines mainly differs in the way they are used in multithreaded contexts; this point will be discussed later, in section 4.9; for the moment, let us consider that there is no differences between synchronized and unsynchronized machines.

A reactive instruction can be given at construction; when it is the case, it becomes the initial program of the machine. Reactions of reactive machines (often simply called "machine", for short) are obtained using the `react` method of interface `Machine`.

One can now gives a minimalist example of executable **Junior** code:

```
import junior.*;
import junior.extended.*;

public class HelloWorld
{
    public static void main(String[] args){
       Machine machine = Jr.SyncMachine(Jr.Atom(new Print("hello, world!")));
       for (int i = 0; i < 3; i++){
           System.out.print("instant "+i+": ");
           machine.react();
           System.out.println("");
       }
    }
 }
```

First, two packages are imported:

- `junior`, to be able to use classes `Machine`, `SyncMachine`, `Jr`, and `Atom`.
- `junior.extended`, for using action `Print`.

Then, a standard Java class is defined with a `main` method. The body of `main` starts by defining a reactive machine with an initial program which prints a message. Finally, the reactive machine is activated 3 times (a trace shows the sequence of machine reactions).

One obtains the output:

```
instant 0: hello, world!
instant 1:
instant 2:
```

The message is printed at the first instant, that is during the first machine reaction. The two next reactions are empty, as the machine program is completely terminated at the end of first reaction.

An other way to define the previous machine would be to drop the program into the machine, using the `add` method:

```
Machine machine = Jr.SyncMachine();
machine.add(Jr.Atom(new Print("hello, world!")));
```

The result would be exactly the same as previously.

The `Stop` instruction is the basic way to delay execution for next instant. Executing a `Stop` terminates execution for current instant; however, execution is not completely terminated at this stage, and the `Stop` instruction becomes the new starting point for next instant. The state associated to `Stop` encodes end of current instant: it changes at the end of the first instant of execution, indicating that instruction is completely terminated. Running the previous machine with program:

```
Jr.Seq(Jr.Atom(new Print("hello, ")),
Jr.Seq(Jr.Stop(),
       Jr.Atom(new Print("world!"))))
```

would produce:

```
instant 0: hello,
instant 1: world!
instant 2:
```

The `Stop` instruction splits the execution in two distinct instants: "`hello,`" is printed during the first one, while "`world!`" is printed during the second one.

2.3 Dynamicity

Dropping a program into a machine can occur at any time. For example, one can trigger the dropping of an instruction by pressing on a button displayed on screen; the following code is a Java1.1 "action listener" to perform this:

```
class Drop implements ActionListener
{
    Machine machine;
    Program program;
    public Drop(Machine machine,Program program){
        this.machine = machine; this.program = program;
    }
    public void actionPerformed(ActionEvent e){
       machine.add(program.copy());
    }
}
```

The `copy` method of `Program` produces new fresh copies of reactive instructions. Thus, a new program is dropped each time `actionPerformed` is called.

Now, here is the code fragment to define a graphical button, which prints a message when pressed:

```
Machine machine = Jr.SyncMachine();
Button button = new Button("press here");
button.addActionListener(
   new Drop(machine,Jr.Atom(new Print("hello, world!\n"))));
Frame frame = new Frame();
frame.add(button);
```

An important point is that without copying the dropped instruction in method `actionPerformed`, only the first press action would print a message, as reactive instructions are not reentrant; indeed, to drop in a machine an instruction which is already terminated, has no effect.

2.4 Events

Event are non persistent data with a binary status *present* or *absent*, possibly changing at each instant. An event becomes present during one instant as soon as it is generated by a program component during this instant. A strong coherency property holds: *during one instant, the same event cannot be tested as present by one component and as absent by another component.* In other words: *events are broadcast.*

A way to implement the coherency property of **Junior** machines is as follows:

- A new *unknown* event status is introduced; the machine assigns it to all events at the beginning of each instant.
- The status of an event is changed to *present* as soon as it is generated.
- When the machine detects that no new event can be generated, it changes to *absent* the status of all unknown events and decides the end of current instant.

Note that end of instant and absence of events are decided together, in the same step; this has important consequence, which will be discussed later in section 4.2.

The **Jr** API gives several ways to deal with events. In the simplest one, events are identified by strings. For example, to generate an event named `e`, one writes: `Jr.Generate("e")` and to wait for it: `Jr.Await("e")`. In this last instruction, control is stopped while event `e` is not generated, and the instruction is completely terminated when `e` becomes present. The `Await` instruction has an associated state which codes for termination, that is, for the awaited event generation.

The following code illustrates event broadcasting:

```
Button button1 = new Button("accumulate");
button1.addActionListener(new Drop(machine,
    Jr.Seq(Jr.Await("event"),Jr.Atom(new Print("hello, world!\n")))));
```

```
Button button2 = new Button("flush");
button2.addActionListener(new Drop(machine,Jr.Generate("event")));
```

So, a new copy of :

```
Jr.Seq(Jr.Await("event"),Jr.Atom(new Print("hello, world!")))
```

is dropped into the machine each time the first button is pressed. When the second button is pressed, a new `Generate` instruction is dropped into the machine, which triggers the previously dropped waiting instructions. The broadcast property of events insures that all waiting instructions are triggered in the same instant: actually, the machine program is empty after pressing the second button.

2.5 Concurrency

Junior owns a concurrency constructor, named `Par` (for parallelism) which puts two reactive instructions A and B in parallel: A and B are executed at each instant, and the parallel construct is completely terminated when both A and B are. The state of `Par` is the union of the state of A and of the state of B. The order in which, at each instant, A and B are executed is left unspecified. Thus, the result of executing:

```
Jr.Par(Jr.Atom(new Print("hello, ")),
       Jr.Atom(new Print("world!")))
```

can be as well "`hello, world!`" or "`world!hello, `".

Reactive instructions dropped in a machine are put in parallel with the machine program. For example:

```
machine.add(Jr.Par(A,B));
```

is strictly equivalent to:

```
machine.add(A); machine.add(B);
```

To simplify programming and reasoning about reactive programs, an instruction dropped into a machine during the course of a reaction is not immediately run by the machine; actual adding of the instruction to the machine program is delayed to the *beginning of the next instant*. Actually, this is quite a general attitude in **Junior**: to avoid interferences, program changes issued by the external world should be systematically delayed to the next instant.

2.6 Interfacing with Java

Interfacing reactive instructions with Java is based on the notion of an *implicit* Java object set by a *link* instruction. The implicit Java object can be directly accessed and transformed by atoms executed by the link body; it is also used by expressions evaluations triggered by reactive instructions and returning values for them.

Implicit Object

The `Jr.Link(object,p)` reactive instruction defines `object` as being the implicit Java object associated to the reactive instruction `p`.

Actions executed by atoms must implement the `execute` method with signature `void execute(Environment env)`. The environment allows the action to get access to the implicit Java object (if defined), using method `linkedObject`. For example, consider:

```
class Incr implements Action
{
    public void execute(Environment env){
       Button b = (Button)env.linkedObject();
       int v = Integer.parseInt(b.getLabel());
       b.setLabel(""+(v+1));
    }
}
```

The implicit object is obtained by the call "`env.linkedObject()`" which returns an instance of type `Object`. Actually, one supposes that the implicit object of `Incr` is always a button of type `Button`; thus, to get access to it, one must cast it to `Button`. Now, in order to cyclically increment the label of `button`, it becomes sufficient to add the following instruction in the reactive machine:

```
Jr.Link(button,Jr.Loop(Jr.Seq(Jr.Atom(new Incr()),Jr.Stop())))
```

Expressions

There are several kinds of expressions used to interface reactive instructions with Java; for example, an integer expression computes the number of iterations a repeat loop has to perform. A boolean expression used by `If` reactive instructions is another example; it must implement method with signature `public boolean evaluate(Environment env)` of class `BooleanWrapper`. The moment evaluation of an expression takes place is under control of the reactive instruction using it. For example, here is the definition of a boolean expression that returns the value of variable `automatic` of its implicit object:

```
class Automatic implements BooleanWrapper
{
   public boolean evaluate(Environment env){
       return ((Robot)env.linkedObject()).automatic;
   }
}
```

2.7 Migration

In **Junior**, it is possible to extract instructions from machines, getting what remains to be executed. This is the inverse of the dynamic dropping of instructions provided by the `add` method of machines. Actually, not all instructions can be extracted, but only *freezable*

ones. Instruction `Jr.Freezable("event",p)` defines `p` as a freezable instruction which can be extracted from the executing machine by generating `event`. The result of the extraction is a new reactive instruction representing what remains to be done by `p` from the point it is frozen, up to complete termination. The extracted instruction is called the *residual* of `p`.

Three points are important:

- Only currently running freezable instructions can be removed from machines.
- The removal of a freezable instruction does not prevent it to react at the very instant the freezing event occurs.
- The residual obtained from a freezable instruction is only available at the instant which immediately follows generation of the freezing event.

Here is, for example, an action which transfers (by calling the static method `Migration.transfer`) the residual of a freezable instruction (freezing event is `freeze`):

```
class Migrate implements Action
{
    public void execute(Environment env){
        Program residual = env.getFrozen(Jr.StringIdentifier("freeze"));
        if (residual != null) Migration.transfer(residual);
    }
}
```

Method `getFrozen` returns the residual associated to an event, if it exists; it returns `null` otherwise.

As a consequence of the previous definition of a freezable instruction, the following actions must be performed in sequence in order to extract a freezable instruction:

1. generate the freezing event;
2. wait for next instant;
3. get the residual associated to the freezing event.

Here is, for example, how to define a button which, when pressed, transfers the residual of a freezable instruction (by running the previous `Migrate` action):

```
button.addActionListener(
   new Drop(machine,
       Jr.Seq(Jr.Generate("freeze"),
       Jr.Seq(Jr.Stop(),Jr.Atom(new Migrate())))));
```

The following method assigns a freezable behavior to a button:

```
public void setButtonBehavior(Program p){
      machine.add(Jr.Link(button,Jr.Freezable("freeze",p)));
}
```

Suppose that there are two buttons and that a cyclic behavior is initially transferred to one of the two buttons, by calling method `transfer` (which in turn calls `setButtonBehavior`):

```
transfer(Jr.Loop(Jr.Seq(Jr.Atom(new Incr()),Jr.Stop())));
```

Now, the result of pressing on the currently running button is to transfer the cyclic behavior to the other button, which thus becomes the new active button.

This little example shows how code migration can be based on the notion of a freezable instruction. Of course, migration through the network needs more: there must be a way to "serialize" reactive instructions, in order to transmit them (serialization is provided for free by Java). One does not consider this question in more details here.

2.8 Preemption and Control

Junior defines two operators to get fine control over reactive instructions; one is a preemption operator which forces a reactive instruction to terminate when an event is present; the other one allows a reactive instruction to execute according to presence of an event.

Preemption

To freeze a reactive instruction preempts it: execution of the frozen instruction is definitely abandoned by the executing machine. However, there is a side-effect which is to produce the residual of the frozen instruction. Instruction `Until` of **Junior** has the same behavior than `Freezable`, but does not computes any residual. It has the form: `Jr.Until("event",body,handler)`, where `body` and `handler` are two reactive instructions. Execution of `body` is abandoned (one says, it is *preempted*) as soon as `event` becomes present; in this case, control directly goes to `handler` which is then executed.

For example, consider:

```
Program controller =
   Jr.Loop(
        Jr.Until("suspend",
               Jr.Loop(Jr.Seq(Jr.Generate("step"),Jr.Stop())),
               Jr.Seq(Jr.Stop(),Jr.Await("resume"))));
```

At each instant, inner loop generates event `step`. It is preempted by event `suspend`. In case of preemption, execution stops, and, at next instant, waiting for `resume` starts. Finally, the `Until` instruction completely terminates when `resume` occurs. As `Until` is enclosed by the external loop, it is then restarted, and cyclic generation of `step` begins again. Thus, `controller` actually implements a two-states behavior:

- In first state, `step` is generated at each instant; transition to second state occurs when `suspend` is generated.
- Execution remains in second state, while `resume` is not generated, and it returns to first state when it is.

Control

The **Junior** `Control` instruction gives a way to execute a reactive instruction only at instants where a given event is present. At others instants, the reactive instruction just stays in same state, without executing anything. For example, in the following program, `body` is controlled by event `step` and is executed only when it is generated:

```
Program controlledBody =
    Jr.Seq(Jr.Control("step",body),Jr.Generate("term"));
```

`Control` is completely terminated when `body` is, and, then, event `term` is generated. Actually, the `Control` instruction can be seen as "filtering" instants[2] for its body: the body can proceed only when `Control` let instants reach it.

3 Junior API

This section describes **Jr**, the **Junior** API (*Application Programming Interface*).

3.1 Identifiers

In section 2, events were identified by strings; however, in **Junior**, the basic way to identify events is more abstract and uses *identifiers*. Identifiers implement the `Identifier` interface which defines equality and hashcode for identifiers. There is a unique assumption: equality of two identifiers of the same object must always return true. Equality and hashcode methods must be implemented as for the basic Java class `Object` (this allows, as example, to store identifiers in hashtables).

```
public interface Identifier extends java.io.Serializable
{
   boolean equals(Object object);
   public int hashCode();
}
```

Here is the implementation of identifiers as strings:

```
public class StringIdentifier implements Identifier
{
    public String id;
    public StringIdentifier(String id){ this.id = id; }
    public int hashCode(){ return id.hashCode(); }
    public String toString(){ return id; }
    public boolean equals(Object object){
       if (object instanceof StringIdentifier)
            return id.equals(((StringIdentifier)object).id);
       return false;
    }
}
```

[2] This defines a kind of logical clock slower than the one of instants.

3.2 Programs

Reactive instructions are manipulated as programs, satisfying the following interface:

```
public interface Program extends java.io.Serializable
{
   Program copy();
}
```

Thus, basically programs can be copied (to get new executable instances; programs are not reentrant) and are serializable, a property useful for migration, as indicated in 2.7

3.3 Machines

A reactive machine is running a program, defining instants for it and performing broadcast of generated events. New programs can be dynamically added to the machine; they are put in parallel with the current machine program. Running program components can be removed from the machine, after being frozen. Reactive machines implement the `Machine` interface:

```
public interface Machine
{
   boolean react();
   void add(Program program);
   void generate(Identifier event);
   void generate(Identifier event, Object val);
   Program getFrozen (Identifier event);
}
```

- A reaction of the machine is performed in response to a call of the `react` method. The call returns true if the machine program is completely terminated, and false otherwise. Machine reactions are defining machine instants of execution.
- The `add` method adds its parameter to the machine program. Added instructions are collected during machine reaction and are incorporated to the machine program at the beginning of the next instant. So, program executions are independent of the actual timing of dropping actions during instants.
- Methods `generate` generate events in the machine. Generation of an event should not occur after decision of the end of instant (which would contradict the fact that it is absent). Values, which are Java objects, can be associated to event generations. Values associated to events are accessible by the environment provided by the machine (described in 3.5)
- Method `getFrozen(event)` returns the program constructed from residuals of instructions frozen (by generating `event`) during previous reaction (using `Freezable` instruction; see 3.11); if several instructions were frozen, residuals are put in parallel in the result of `getFrozen`. If no instructions were frozen during previous reaction, `getFrozen` returns the `null` value.

Actually, machines are implemented in two ways: synchronized machines and unsynchronized machines.

Synchronized Machines

Methods of synchronized machines are protected against concurrent accesses, which makes programming simpler in multithreaded contexts (for example, when the network is involved). Moreover, events generations comming from the external world are, like additions of new programs, systematically delayed to the next instant. More precisely, protection given by synchronized machines means the following:

- Methods are protected against access by concurrent threads. For example, two threads can concurrently add new programs without interference.
- Event generations issued by calls of `generate` are collected during current machine reaction, and become actual at the beginning of next instant. In this way, there is no risk that generation of an event occurs after the end of instant.

Empty interface `SyncMachine` flags synchronized machines:

```
public interface SyncMachine {} extends Machine
```

The constructors for synchronized machines are:

```
SyncMachine Jr.SyncMachine(Program p)
SyncMachine Jr.SyncMachine()
```

In the first constructor, the argument program is the initial machine program. In the second, the initial program is `Jr.Nothing()`.

Unsynchronized Machines

With unsynchronized machines, there is no waranty against methods interferences; protection of methods is thus under programmers responsability. Moreover, event generations are immediately performed by the machine; programmers must thus ensure that an event generation cannot occur after end of instant decision. Empty interface `UnSyncMachine` flags unsynchronized machines:

```
public interface UnSyncMachine {} extends Machine
```

The constructors for unsynchronized machines are:

```
UnSyncMachine Jr.UnSyncMachine(Program p)
UnSyncMachine Jr.UnSyncMachine()
```

3.4 Basic Programs

The basic **Junior** programs are:

- `Nothing` does nothing and immediately terminates.
- `Stop` does nothing, stops for the current instant, and terminates at next instant.
- `Seq` puts a program in sequence with an other. Execution of the second program is immediately started when the first program becomes completely terminated.
- `Par` puts two programs in parallel. The parallel instruction is completely terminated when its two left and right branches are also completely terminated.
- `Loop` defines an infinite loop. Execution of the body is immediately restarted from its beginning when it is completely terminated.

Corresponding constructors are:

```
Program Jr.Nothing()
Program Jr.Stop()
Program Jr.Seq(Program first, Program second)
Program Jr.Par(Program left, Program right)
Program Jr.Loop(Program body)
```

It is the programmer responsability to get control over loops that never converge, executing their body during one instant without never ending. These loops are called *instantaneous loop*; executing an instantaneous loop would prevent other program component to be executed.

3.5 Environments

Environments allows atoms and expressions to access values associated to generated events, residuals of frozen instructions, and implicit object. Environments are parameters of `execute` and of `evaluate` methods of actions and expressions. Interface `Environment` is:

```
public interface Environment
{
   Object[] previousValues (Identifier id);
   Object[] currentValues (Identifier id);
   Program getFrozen (Identifier id);
   Object linkedObject();
}
```

- `previousValues` returns the array of values generated for one event *during previous instant*. These values are only available during current instant (that is, the instant following actual generations), and are lost later. The `null` value is returned if no valued generation has occured during previous instant; otherwise, values are available with two special cases:

  - `Jr.NO_VALUE` indicates a generation without value.
  - `Jr.NULL_VALUE` indicates a generation with the `null` value.

- `currentValues` returns the array of values generated for one event *during current instant*. The same conventions are adopted as for `previousValues`.
- `getFrozen` returns the same information than the corresponding method of `Machine` (see 3.3).
- `linkedObject` returns the implicit object, set by `Link` instructions (see 3.10). The `null` value is returned in case there is no associated implicit object.

3.6 Actions

Actions are used by atoms to interact with Java in an atomic manner. Interface `Action` is defined by:

```
public interface Action extends java.io.Serializable
{
   public void execute(Environment env);
}
```

Constructor for atoms is:

```
Program Jr.Atom(Action action)
```

As example, here is the definition of the `Print` action used to trace execution (this action does not actually belong to package `junior`):

```
public class Print implements Action
{
  public String msg;
  public Print(String msg){ this.msg = msg; }
  public String toString(){ return "System.out.print(\""+msg+"\")"; }
  public void execute(Environment env){ System.out.print(msg); }
}
```

3.7 Wrappers

Wrappers return a value when their `evaluate` method is called. They give possibility to reactive instructions to compute values during execution (and not only at construction). For example, an integer wrapper (with an `evaluate` method returning an integer) is used by the `Repeat` finite loop instruction to compute the number of iterations of the body. There is a general rule: wrappers are evaluated (that is, the `evaluate` method is called) just before executing for the first time the reactive instruction in which they are embedded.

Basic Wrappers

There are four interfaces for wrappers returning identifiers, booleans, integers, or general Java objects. In all cases, the `evaluate` method has the machine environment as parameter.

```
public interface IdentifierWrapper extends java.io.Serializable
{
   Identifier evaluate(Environment env);
}


public interface BooleanWrapper extends java.io.Serializable
{
   boolean evaluate(Environment env);
}


public interface IntegerWrapper extends java.io.Serializable
{
  public int evaluate(Environment env);
}


public interface ObjectWrapper extends java.io.Serializable
{
   Object evaluate(Environment env);
}
```

Constant Wrappers

A constant wrapper always returns the same value given at construction. There are 4 constructors for constant wrappers:

```
IdentifierWrapper Jr.ConstWrapper(Identifier id)
ObjectWrapper Jr.ConstWrapper(Object obj)
IntegerWrapper Jr.ConstWrapper(int n)
BooleanWrapper Jr.ConstWrapper(boolean b)
```

3.8 If

At first instant of execution, `If` reactive instruction evaluates a boolean wrapper, and, accordingly to the result, chooses the "then" or "else" branch for execution. Execution may take more than one instant, and at next instants, the chosen branch is continued. Constructors are:

```
Program Jr.If(BooleanWrapper cond, Program thenInst, Program elseInst)
Program Jr.If(BooleanWrapper cond,Program thenInst)
```

The second constructor is a syntactic facility: `Jr.If(cond,thenInst)` is equivalent to instruction `Jr.If(cond,thenInst,Jr.Nothing())`.

3.9 Repeat

`Repeat` reactive instructions implement finite loops. The number of iterations is returned by an integer wrapper which is evaluated at the first instant of execution, before body execution. There are two constructors:

```
Program Jr.Repeat(IntegerWrapper wrapper, Program body)
Program Jr.Repeat(int count, Program body)
```

The second constructor is a syntactic facility: `Jr.Repeat(n,body)` is equivalent to instruction `Jr.Repeat(Jr.ConstWrapper(n),body)`. It is used when the number of iteration is a constant.

3.10 Links

A link associates a Java object to a reactive statement; the object becomes the implicit Java object (returned by method `linkedObject` of the environment; see 3.5) while executing the instruction. There are two constructors:

```
Program Jr.Link(ObjectWrapper wrapper, Program body)
Program Jr.Link(Object object, Program body)
```

The second constructor is an abreviation: `Jr.Link(obj,body)` is equivalent to instruction `Jr.Link(Jr.ConstWrapper(obj),body)`. It is used when `obj` is determined at construction.

3.11 Freezable

A freezable instruction executes its body, while a freezing event is not generated. In case the event is generated, then the body is extracted from the program, after having finished to react for current instant, and the residual (what remains to do for future instants) is stored in the environment. The residual can be obtained at next instant, by calling method `getFrozen` of the environment (for atoms; see 3.5) or of the machine (for interfacing with Java; see 3.3).

```
Program Jr.Freezable(IdentifierWrapper wrapper, Program body)
Program Jr.Freezable(Identifier event, Program body)
```

The second constructor is an abreviation: `Jr.Freezable(event,body)` is equivalent to `Jr.Freezable(Jr.ConstWrapper(event),body)`. It is used when `event` is determined at construction.

3.12 Generate

An event is a data which is present or absent during instants. An event becomes present during one instant as soon as it is generated during this instant; otherwise, it is absent. Events are broadcast: during the course of an instant, the same event cannot be tested

with two different presence status (present or absent). Values can be associated to event generations: they are collected during current instant and are available through the machine environment (see 3.5).

Generate without value

Event generations without values are managed by the following constructors:

```
Program Jr.Generate(IdentifierWrapper iwrap)
Program Jr.Generate(Identifier event)
```

The second constructor is an abreviation: `Jr.Generate(event)` is equivalent to instruction `Jr.Generate(Jr.ConstWrapper(event))`. It is used when `event` is determined at construction.

Generate with value

There are 4 constructors for generations with values:

```
Program Jr.Generate(IdentifierWrapper iwrap,ObjectWrapper owrap)
Program Jr.Generate(Identifier event,ObjectWrapper owrap)
Program Jr.Generate(IdentifierWrapper iwrap,Object value)
Program Jr.Generate(Identifier event,Object value)
```

The three last constructors are abreviations, mapping values to wrappers in the standard way; for example: `Jr.Generate(event,value)` is equivalent to:

```
Jr.Generate(Jr.ConstWrapper(event),Jr.ConstWrapper(value))
```

3.13 Configurations

Configurations are boolean combinations of events, used by some reactive instructions (for example, `Await`, defined in 3.14). There are 4 kinds of configurations:

- `Presence` is the basic configuration, associated to an event. It is satisfied when the event is present, and unsatisfied when it is absent, that is when the current instant is terminated.
- `And` is the conjunction of two configurations; it is satisfied when both are statisfied, and unsatisfied when at least one is unsatisfied.
- `Or` is the disjunction of two configurations; it is satisfied when one (or both) is statisfied, and unsatisfied when both are unsatisfied.
- `Not` is the negation of a configuration; its is satisfied when the configuration is unsatisfied, and unsatisfied when the configuration is satisfied.

Configurations are used through interface:

```
public interface Configuration extends java.io.Serializable
{
   Configuration copy ();
}
```

Constructors for configurations are:

```
Configuration Jr.Presence(IdentifierWrapper wrapper)
Configuration Jr.Presence(Identifier event)
Configuration Jr.And(Configuration c1, Configuration c2)
Configuration Jr.Or(Configuration c1, Configuration c2)
Configuration Jr.Not(Configuration c)
```

The second constructor is an abreviation: `Jr.Presence(event)` is equivalent to instruction `Jr.Presence(Jr.ConstWrapper(event))`. It is used when `event` is determined at construction.

3.14 Await

The `Await` instruction waits for a configuration to be satisfied; the waiting can take several instants; execution is stopped while the configuration is unsatisfied and it terminates when it is satisfied (of course, it can never terminate if the configuration is never satisfied).

```
Program Jr.Await(Configuration config)
Program Jr.Await(IdentifierWrapper wrapper)
Program Jr.Await(Identifier event)
```

The second and third constructors are abreviations: `Jr.Await(wrapper)` is equivalent to instruction `Jr.Await(Jr.Presence(wrapper))`, and `Jr.Await(event)` is equivalent to `Jr.Await(Jr.Presence(event))`.

3.15 Until

The `Until` instruction preempts its body when a configuration becomes satisfied. In case of preemption, execution of the body is complete for current instant, but is abandonned for future instants; moreover, in this case, execution switches to the handler part of the `Until` instruction.

```
Program Jr.Until(Configuration config, Program body, Program handler)
Program Jr.Until(Configuration config, Program body)
```

The second constructors is an abreviation: `Jr.Until(conf,body)` is equivalent to instruction `Jr.Until(conf,body,Jr.Nothing())`.

Identifier wrappers and identifiers can be used in place of configurations:

```
Program Jr.Until(IdentifierWrapper wrapper, Program body, Program handler)
Program Jr.Until(Identifier event, Program body, Program handler)
```

These constructors are abreviations, mapping values to wrappers in the standard way; for example: `Jr.Until(wrapper,body,handler)` is equivalent to:

```
Jr.Until(Jr.Presence(wrapper),body,handler)
```

Finally, as previously, the handler part can be omitted, and in this case it is the instruction `Jr.Nothing()`. This gives the two constructors:

```
Program Jr.Until(IdentifierWrapper wrapper, Program body)
Program Jr.Until(Identifier event, Program body)
```

3.16 When

The `When` instruction is similar to `If` except that an event configuration is tested instead of a boolean expression. First, `When` evaluates a configuration, and if it is satisfied, execution goes to the "then" branch; otherwise, execution goes to the "else" branch. An important point is that evaluation of a configuration can take the entire instant if the absence of some events is to be considered. In this case, execution of the chosen branch can only start at the instant that follows configuration evaluation.

```
Program Jr.When(Configuration config, Program thenInst, Program elseInst)
```

Identifier wrappers and identifiers can be used in place of configurations:

```
Program Jr.When(IdentifierWrapper wrapper,Program thenInst,Program elseInst)
Program Jr.When(Identifier event, Program thenInst, Program elseInst)
```

These constructors are abreviations, mapping values to wrappers in the standard way.

The "else" part can be omitted, and in this case it is `Jr.Nothing()`. This gives constructors:

```
Program Jr.When(Configuration config,Program thenInst)
Program When(IdentifierWrapper wrapper,Program thenInst)
Program When(Identifier event, Program thenInst)
```

3.17 Control

The body of a `Control` instruction is executed only at instants where a controlling event is present.

```
Program Jr.Control(IdentifierWrapper wrapper, Program body)
Program Jr.Control(Identifier event, Program body)
```

The second constructor is an abreviation: `Jr.Control(event,body)` is equivalent to instruction `Jr.Control(Jr.Presence(event),body)`.

3.18 Local

The `Local` instruction defines a local event in the scope of its body:

```
Program Jr.Local(Identifier event, Program body)
```

3.19 Events identified by Strings

Events can be systematically denoted by Java strings. Denotation is based on the following functions:

- `String` to `Identifier`: the `StringIdentifier` function defined in 3.1 transforms strings into identifiers.
- `String` to `IdentifierWrappers`: the `StringWrapper` function transforms strings into identifier wrappers; it is defined by:

```
IdentifierWrapper StringWrapper(String str){
   return Jr.ConstWrapper(StringIdentifier(str));
}
```

- `String` to `Configuration`: the following `StringConfig` function transforms strings into configurations.

```
Configuration StringConfig(String str){
   return new Presence(StringWrapper(str));
}
```

Using the previous functions, one has the new constructors:

```
Program Jr.Generate(String event,Object val)
Program Jr.Generate(String event)
Program Jr.Await(String event)
Program Jr.Control(String event, Program body)
Program Jr.Local(String event, Program body)
Program Jr.Freezable(String event, Program body)
Program Jr.Until(String event, Program body, Program handler)
Program Jr.Until(String event, Program body)
Program Jr.When(String event,Program thenInst, Program elseInst)
Program Jr.When(String event,Program thenInst)
Configuration Jr.Presence(String event)
Configuration Jr.Not(String event)
```

4 Details of Junior Programming

This section contains a discussion on several aspects of **Junior** programming.

4.1 Identifiers

In **Junior**, events are, in an abstract way, denoted by identifiers (see 3.1). The reason to choose this abstract vision is that it makes easier interfacing with a distributed execution platform; actually **Junior** is interfaced with the ORB Jonathan[9], which has its own notion of an identifier. Abstract **Junior** identifiers allows one to directly map them on Jonathan identifiers.

To make programming easier, **Jr** gives several means for building events. With the simplest one, events are identified by Java strings, and several methods are provided to build identifier wrappers and configurations out of strings.

4.2 Reaction to Absence

In **Junior**, an event cannot be considered as absent before the end of the instant. This is a coherent point of view as, otherwise, the event could be generated after being considered as absent; it would thus be absent and present during the same instant, a situation that would contradict the definition of an event. A different point of view is the one of Synchronous Languages[5], for example Esterel[1], in which events (called *signals*) can be considered as absent in the course of an instant, provided that a static analysis of the program, performed at compile time, has indicated that no generation of the event remains possible. **Junior** chooses not to uses static analysis technics; this is the reason why, in **Junior**, one has to wait for the end of instant to be sure that events are really absent. There are several consequences:

- Instructions in sequence with an absence test are always postponed to the next instant. For example, consider:

  ```
  Jr.Seq(Jr.When("event",Jr.Nothing()),Jr.Generate("other"))
  ```

  Consider the case where `event` is absent. Then, as the test of absence "takes the entire current instant", `other` cannot be generated during the current instant, but it is during the next. Situation would be exactly the same with:

  ```
  Jr.When("event",Jr.Nothing(),Jr.Generate("other"))
  ```

- To determine the status of a configuration (satisfied or unsatisfied) can also "take the entire instant", if absent events are to be considered. For example, in instruction `Jr.Await(Jr.Not("event"))` termination occurs at the instant following the one where `event` is absent.

Consequences concerning the `Until` instruction are considered in 4.3.

4.3 Preemption

The `Until` and `Freezable` instructions always let their body react, even if the triggering event (the preemption event for `Until`, and the freezing event for `Freezable`) is present[3]. In some cases, this can have consequences for instruction termination; for example, consider:

```
Jr.Until("trigger",Jr.Loop(Jr.Stop()))
```

Execution is stopped while `trigger` is not generated, and `Until` completely terminates when `trigger` is generated. The point is that, in this case, generation and termination both occur at the same instant. The situation is different with:

```
Jr.Until("trigger",Jr.Await("never"))
```

Let us suppose that `never` is absent and `trigger` present; then, as `trigger` is present, preemption occurs; but to determine that `never` is absent takes the entire instant, and, thus, complete termination of `Until` is postponed to the next instant. To know the exact instant of instructions termination may be a difficult task in some situations; this is certainly one of the difficulties of **Junior** programming.

4.4 Reactivity

Convergence

The basic assumption of **Junior** is that execution during an instant always converges and leads to a stable state from which next instant can start. There are two possibilities for the assumption to be violated:

- Executing an action which never converge will forbid the machine to finish current instant. These situations are out of control of the reactive program and it is the programmer responsability to avoid them. Here is an example of "pathological" action:

  ```
  public class NeverEnding implements Action
  {
    public void execute(Environment env){ while(true){/* empty */} }
  }
  ```

  One could think of machine implementations that would force action termination according to some criteria (as, for example, a fixed delay givent for each action execution). Present implementations do not consider this aspect, which is left for further research.

- Instantaneous loops are loops with immediately terminating bodies (see 3.4). Instantaneous loops have to be eliminated. In **Junior**, it is the programmer responsability to control this point. A way to do this, could be to extend class `Loop` and to systematically put a `Stop` instruction in parallel with the loop body. This is the standard approach of SugarCubes, in which instantaneous loop elimination is the default behavior.

[3] One says that the preemption of `Until` is "weak", compared to the "strong" one of the `abort` construct of Esterel.

Note that there is no possibility to fall in a non-convergent situation by recursive addings of programs, as additions are systematically postponed to the next instant.

Response Time

In addition to convergence of reactive programs, one sometimes want to be sure that instants will not take too much time. That is, one looks for a kind of *response time waranty*. This is presently out of scope in **Junior**. We plan to investigate on this subject; more precisely, the idea would be twofold:

- To restrict actions to a subset of Java, in which one could measure (or control) the execution time of statements. Or course, this would be an extremely severe limitation to the language.
- To produce, when possible, finite state machines, called *automata*, from **Junior** programs.

Analyzing automata produced from programs with restricted actions, one could determine the maximum transitions time of automata, which could be an element for answering the response time problem for reactive programs.

4.5 Events and Values

Values can be associated to event generations. All values associated to one event are collected during the instant, and they are available at next instant through the environment (`previousValues` method; see 3.5).

There are two difficulties, using generations with values:

- Values are returned with type `Object`; it is the programmer responsability to make the correct cast, when it is needed. The reflexive facilities of Java could help in using these values correctly.
- The order in which values are collected depend on the order `Par` instructions choose their branches. It is thus not possible to make any assumption on it.

Junior also provides a way to get values associated to event generations performed during current instant (`currentValues` method; see 3.5). To access values during current instant is extremely error prone; for example, the fact that `currentValues` returns a `null` value does not mean that event is absent, because it may be generated later during the instant.

4.6 Implementation of Parallelism

The `Par` operator does not specify order in which branches are executed; this introduces non-determinism in program behaviors as program semantics depends on the order `Par` branches are executed. Actually, presence of operators such as `Par` is rather natural in the context

of parallel programming, because they are reflecting nondeterminism associated to parallelism. However, it is possible for `Par` implementations to always choose the same fixed order; the benefit is then to reduce nondeterminism of program. This is the option taken by two existing implementations (called REWRITE and REPLACE). However, in a third implementation (called SIMPLE), the order of parallel branches execution can change, depending on the actual programs executed. Thus, Junior programmers should not rely on the order `Par` branches are executed.

In REWRITE and REPLACE implementations, `Par` executes its two branches in a fixed order; more precisely, at each instant, `Par(A,B)` first starts executing the first branch `A`, then the second one `B`. For example, consider:

```
Jr.Par(Jr.Atom(new Print("hello, ")),
       Jr.Atom(new Print("world!")))
```

In REWRITE and REPLACE implementations, the only possible output is message "`hello, world!`".

An important point is that fixing the execution order of `Par` branches does not mean that final effect of the first branch occurs before final effect of the second branch. For example, consider:

```
Jr.Par(
      Jr.Seq(Jr.Await("e"),Jr.Atom(new Print("hello, "))),
      Jr.Seq(Jr.Generate("e),Jr.Atom(new Print("world!"))))
```

Actually, execution is as follows:

- Control is suspended on the `Await` instruction, as `e` is not yet generated.
- Then, control goes to the second branch, and `e` is generated; as the `Generate` instruction is completely terminated, "`world!`" is printed, which terminates execution of the second branch.
- Control returns to the first `Par` branch (as it was suspended), and, because `e` is generated, the `Await` instruction terminates and "`hello, `" is printed.

Thus, synchronization introduced by event `e` has the effect of inverting the two printing actions.

4.7 Synchronized Machines

There is no need for a synchronized machine when an executable program is run by one unique thread. However, there are many cases where several execution threads are used: for example, Java applets create several threads to process graphics and graphical events; concurrent threads also naturally appear when network is involved.

Let us return to example of section 2.3; the machine should be synchronized to avoid uncontroled interferences of buttons press actions with program execution. Moreover, as the Java graphical toolkit Awt is involved, it is mandatory to give the graphical thread enough time to work. This leads to the following executable code:

```
public class Dynamic
{
    public static void main(String[] args){
       Machine machine = Jr.SyncMachine();
       Button button = new Button("press here");
       button.addActionListener(new Drop(machine,
                     Jr.Atom(new Print("hello, world!\n"))));
       Frame frame = new Frame();
       frame.add(button);
       frame.pack(); frame.show();
       while (true){
          machine.react();
          try{ Thread.sleep(100); }catch(Exception e){}
       }
    }
 }
```

The `Thread.sleep` method is called to give enough time to the graphical thread; value of the parameter is actually dependent of the execution machine.

4.8 Reactive Instructions vs Threads

By getting fine control over threads execution, programmers can define their own threads scheduling algorithms. This leads to a flexible and *reflexive* approach to concurrency, in which the basic execution scheme of the system is under control of it. Unfortunately, to get fine control over thread execution while staying in a safe programming context appears to be an extremely difficult task in Java. In first versions, Java provided programmers with 3 primitives to kill, suspend, and resume threads (`Thread.stop`, `Thread.suspend`, and `Thread.resume`). It has been discovered that these primitives are highly error prone, and they have been removed from later versions of the language[8]. A comparison of Java threads with reactive instructions (in the particular context of SugarCubes) can be found in [4]. Here, one limits discussion to the specific problem of gaining fine control over instructions, the one that causes troubles with Java threads.

Using `Until` and `Control`, programmers can get fine control over reactive instructions execution. For example, one can easily define a kind of "reactive thread" which can be started, stopped, suspended, and resumed, as Java threads are (remember that in Java `Thread.stop` kills threads). Actually, reusing the two programs `controlledBody` and `controller` defined in 2.8, the behavior of a reactive thread can be simply defined as:

```
Jr.Seq(Jr.Await("start"),
       Jr.Local("step",Jr.Local("term",
              Jr.Until(Jr.Or(Jr.Presence("stop"),Jr.Presence("term")),
                       Jr.Par(controlledBody,controller)))))
```

Local events are defined using the `Local` instruction. One uses here the full version of `Until`, in which preemption is governed by a *configuration of events*; here, configuration is an `Or` which is satisfied when `stop` or `term` are present (or both).

The behavior of a reactive thread is thus:

- The thread starts when event `start` is generated.
- The thread completely terminates when its body terminates (then, event `term` is generated), or when the thread is stopped (event `stop` is generated).
- The thread is suspended (generations of the local event `step` are stopped) when `suspend` is generated; it is resumed when event `resume` is generated (generations of `step` restart).

This example shows that reactive instructions of **Junior** are actually an alternative to Java threads for concurrent programming when fine control over execution is needed.

4.9 More Friendly Syntax

As already mentionned, **Junior** programs syntax is not very friendly. Several solutions are possible to deal with this situation; for example, one could define a language that would be translated into **Junior**. This is the approach taken by Reactive Scripts[2]. With the syntax of Reactive Scripts (which is actually the one obtained in present implementations when printing programs), the program of reactive threads defined in 4.8 gets a more readeable form:

```
await start;
event step, term in
   do
      control body by step;
      generate term
   ||
      do
         loop generate step; stop end
      until suspend
      handle
         stop; await resume
      end
   until stopped or term
end
```

It is planned to provide users with a translator from a similar syntax to **Junior**.

5 Conclusion

Junior is set of Java classes for reactive programming. It is used through an API which defines a small set of constructors, with a precise semantics. Junior defines a set of high-level primitives for defining reactive programs, such as parallelism and preemption. This leads to a programming which is flexible and which provides users with a fine control over program executions.

The formal semantics of Junior and several existing implementations are described in companion papers which can be found at [10].

Contents

References

[1] G. Berry, G. Gonthier, *The Esterel Synchronous Language: Design, Semantics, Implementation*, Science of Computer Programming, 19(2), 87-152, 1992.

[2] F. Boussinot, L. Hazard, *Reactive Scripts*, Proceedings of the 3rd International Workshop on Real-Time Computing Systems and Applications (RTCSA'96), pp. 270-277, Seoul, IEEE 0-8186-7626-4, 1996.

[3] F. Boussinot, J-F. Susini, *The SugarCubes Tool Box - A reactive Java framework*, Software Practice & Experience, 28(14), 1531-1550, 1998.

[4] F. Boussinot, J-F. Susini, *Java threads and SugarCubes*, Software Practice & Experience, 30(5), 545-566, 2000.

[5] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Pub., 1993.

[6] D. Harel, A. Pnueli, *On the Development of Reactive Systems*, in K. R. Apt (ed.) Logics and Models of Concurrent Systems, NATO ASI Series F, Vol. 13, pp. 477-498, Springer-Verlag, New York, 1985.

[7] L. Hazard, J-F. Susini, F. Boussinot, *The Junior reactive kernel*, Inria Research Report 3732, July 1999.

[8] JavaSoft, *Why JavaSoft is Deprecating Thread.stop, Thread.suspend and Thread. resume*, JavaSoft Documentation, available at:

http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html.

[9] Information on Jonathan is available at http://www.objectweb.org

[10] http://inria.fr/mimosa/rp/junior

Index

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Lorraine : Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot St Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, B.P. 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399